Computational Models - Lecture 10
Fall 04/05

- Undecidability of Tiling (Domino) Problems
- Self Reproducibility
- The S_{mn} and Recursion Theorems.
- Example and Applications: Quines and Viruses
- Unrestricted Grammars and TMs
-
- HoHo Hi Hi (?)

- Sipser, Sections 5.2, 6.1 (partial cover)
- Hopcroft and Ullman, Section 8.8

Undecidability of Tiling Problems

An old times board-and-chalk presentation.

Self-Reproducibility

- Can a system reproduce itself?
- A mechanism that produces some products is usually more complex than those products.
- On the other hand, some biological systems (*e.g.* living cells) seem capable of self-reproduction.
- What about the code of Turing machines, or programming languages in general? Can it reproduce itself?
- Will prove there are self reproducing codes, then give examples.

Preliminaries

- Easy to tell when a string T equals the encoding $\langle M \rangle$ of some TM.
- Given i, can find the lexicographically i-th string among all the encodings of TMs.
- Will denote by f_i the function computed by the TM with the i-th encoding.

The $S_{m,n}$ Theorem

Theorem: Let $s : \Sigma \times \{\sharp\} \times \Sigma \longrightarrow \Gamma$ be a computable function of two variables (possibly partial). Then there is a total computable function of one variable, $\varphi(\cdot)$, such that for all m and n,

$$s(m,n) = f_{\varphi(m)}(n) \ .$$

Recall that $f_{\varphi(m)}(\cdot)$ is the one variable function computed by the TM with the $\varphi(m)$-th encoding.

The $S_{m,n}$ Theorem

Proof: Let S be a TM that computes $s(\cdot,\cdot)$. Consider the following TM, A:
Given input x, A constructs a TM, N_x.
On input y, N_x shifts y to the right by $|x|+1$ squares, writes $x\sharp$ at the beginning of the tape, and then runs S on $x\sharp y$ to obtain $s(x,y)$.

The output of A is the index of the encoding, $\langle N_x \rangle$. Given S, this index is a function of x alone. Denote this index by $\varphi(x)$. Then clearly $\varphi(\cdot)$ is a total, computable function, and for all m and n,

$$s(m,n) = f_{\varphi(m)}(n) \ .$$

The Recursion Theorem

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There exist an index x_0 such that for all y,

$$f_{x_0}(y) = f_{\varphi(x_0)}(y) \ .$$

Such x_0 is called a fixed point of φ. The function φ can be viewed as modifying TMs descriptions.

The Recursion Theorem

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There exist an index x_0 such that for all y,

$$f_{x_0}(y) = f_{\varphi(x_0)}(y) \ .$$

Suppose the modification is that if i-th machine outputs k on input y, then $\varphi(i)$-th machine outputs $k+1$ on input y.

Q.: How then can $f_{x_0}(y) = f_{\varphi(x_0)}(y)$?

A.: If $f_{x_0}(y) = \perp$ everywhere, then so does $f_{\varphi(x_0)}(y)$!

The Recursion Theorem: Proof

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There is an index x_0 such that for all y, $f_{x_0}(y) = f_{\varphi(x_0)}(y)$.

- For each integer i construct a TM, M, that on input y computes $f_i(i)$.
- Then M runs the $f_i(i)$-th TM on y.
- Let $g(i)$ be the index of M's encoding.
- For all i and y, $f_{g(i)}(y) = f_{f_i(i)}(y)$.
- Notice that $g : \mathcal{N} \longrightarrow \mathcal{N}$ is a total computable function, even if $f_i(i)$ is not defined (in such case $f_i(i)$ on y is not defined).

The Recursion Theorem: Proof

Theorem: Let $\varphi : \mathcal{N} \longrightarrow \mathcal{N}$ be a total recursive function, thought of as a function from TM indices to TM indices. There is an index x_0 such that for all y, $f_{x_0}(y) = f_{\varphi(x_0)}(y)$.

- Let j be the index of the encoding of the TM that computes $\varphi \circ g$.
- The index j refers to a TM that on input i, computes $g(i)$, then $\varphi(g(i))$, namely $f_j = \varphi \circ g$.
- Take $x_0 = g(j)$, then $f_{x_0}(y) = f_{g(j)}(y) = f_{f_j(j)}(y) = f_{\varphi(g(j))}(y) = f_{\varphi(x_0)}(y)$.
- So x_0 is a fixed point of the mapping φ, namely the TMs with indices x_0 and $\varphi(x_0)$ compute the same function. $\clubsuit$.

Self Reproducing Code

Theorem: There is an index `self` such that the TM whose index equals `self` outputs `self` on all inputs (including the empty input).

Stated in terms of our "daily programming languages", this corresponds to a program that generates a copy of its own source text as its complete output, without inputting its source.

Such program is known as a quine, after the logician Willard V. Quine (a term suggested by Douglas Hofstadter). Devising the shortest possible quine in some programming language is a common hackish amusement.

Self Reproducing Code

Theorem: There is an index `self` such that the TM whose index equals `self` outputs `self` on all (including the empty) inputs.

Proof: Let φ be the following total computable function from indices to indices of TMs: Given index x, machine with index $\varphi(x)$ ignores input, prints x".

By the recursion theorem, there is an index x_0 such that $f_{x_0} = f_{\varphi(x_0)}$. The function $f_{\varphi(x_0)}$ is computed by a TM, M, that ignores its input and prints x_0. The function f_{x_0} is computed by a TM whose index is x_0.

$f_{x_0} = f_{\varphi(x_0)} \Longrightarrow \; M$ (whose index is x_0) prints x_0 on any input. Taking $\texttt{self} = x_0$, we get a program that prints its own index ("self reproducing code").

Self Reference in English

Print out two copies of the following, the second one in quotes:

> "Print out two copies of the following, the second one in quotes:"

- Part B is clause Print out two copies of the following, the second one in quotes:
- Part A is same with quotes.

Self Reference in General

Make two copies of the following, then translate the first into the language spoken on the other side of the English Channel.

> Faîtes deux copies du suivant, and traduisez le premier en la langue parlé sur l'autre côté de la Manche.

Well, whatever (quoi que ce soit).

Self Reference in Prog. Languages

Scheme:

```
((lambda (x) (list x (list 'quote x)))
'(lambda (x) (list x (list 'quote x))))
```

(try it on your copy of Dr. Scheme.)

It's relatively easy to write quines in other languages such as Postscript which readily handle programs as data; much harder (and thus more challenging!) in languages like C which do not.

Unrestricted Grammars

- $S \longrightarrow ACaB$
- $Ca \longrightarrow aaC$
- $CB \longrightarrow DB$
- $CB \longrightarrow E$
- $aD \longrightarrow Da$
- $AD \longrightarrow AC$
- $aE \longrightarrow Ea$
- $AE \longrightarrow \varepsilon$

What language is generated by this grammar?

Let us switch to David Galles slides (USF)